Supporting Distributed Aspects by Extending Object Teams Model into Distributed Environments

Abdullah O. Al-Zaghameem
Institute of Software Techniques and Theoretical Informatics
Technical University of Berlin
Berlin, Germany
Email: aoztub@cs.tu-berlin.de

***Abstract*—Several distributed AOP models and languages have been developed to support aspects in distributed programming. These approaches prosper in encapsulating distributed concerns within aspect modules and facilitate their employment in distributed applications, but lack supporting management facilities; the dynamic activation/deactivation of aspects at runtime as an alternative to the expensive weaving/unweaving mechanism adopted in some approaches. Additionally, these approaches do not regard the real-world semantics of aspects; which reduces their understandability. In this paper we present DOT/J, a distributed model that extends the programming model of OT/J language into distributed environments to support the dynamic management of these aspects, and improve their semantical representation.**

***Index Terms*—Computer and Information Processing, Distributed Computing, Middleware.**

I. Introduction

The Aspect Oriented Programming technique (AOP)[5] has undoubtedly improve applications development process through separating their different concerns and encapsulate the crosscutting ones in so-called *Aspect* modules. Therefore, applications modularity is highly increased; which makes them easy to implement, understood, and maintain. The brilliant success of AOP technique encourages the employment of aspects in distributed applications. Consequently, several AOP approaches, like Spring AOP[13], extended the structure of their models to support distributed aspects. Some other new distributed AOP models have been developed to mainly cope with distributed aspects, like AWED[7], DyMAC[2], and DJcutter[9].

Most of these distributed AOP models introduce a pointcut-advice model; like the Join-Point Model (JPM) of AspectJ[4] programming language, where crosscutting concerns are defined by coupling pointcuts and advices within aspect modules. A pointcut is a predicate expression that matches specific points of program execution (called join points), and an advice is the action to be taken at a join point matched by a pointcut. Distributed AOP approaches augment this JPM by introducing remote pointcuts and remote advice-execution strategies with various degrees of transparency regarding join-point detection and aspects deployment.

On the one hand, these approaches enable a seamless representation and deployment of distributed aspects but lack supporting dynamic management of these aspects at runtime; like aspect activation/deactivation facility as an alternative to dynamic weaving/unweaving of aspects at runtime which may unintentionally affect application consistency. Additionally, modularizing the context at which these aspects are applied is neglected in most of these approaches; which decreases application modularity and semantics.

On the other, Object Teams/Java (OT/J) [12] is a new programming language that improves the modularity of object-oriented collaboration-based applications by extending Java^{TM}programming language with two new modules; team module and role module. While the first is capture the collaboration at which application's objects are interacting, the second captures the crosscutting behaviors of these objects in collaborations. From an AOP point of view, the seamless separation between application's objects and their collaborative behaviors makes it possible to establish a mechanism to control and manage aspects (or roles) dynamically at runtime, as well as provides a module that captures the context where these aspects are employed.

The rigorous, but flexible, model of OT/J (called OT) motivates us to extend its concepts into distributed environments to support dynamic distributed-aspect management and other distributed programming facilities, such as; distributed components implementation, and adaptable distributed applications development.

This paper is organized as follows: in Section II a quick overview to OT/J programming language concepts will be presented. Section III explains why OT/J doesn't support distributed aspects, and introduces our extending model that invents a new modular and transparent remote role-playing mechanism. Then explains distributed aspect implementation and how to support them dynamically at runtime. Section IV depicts a simple case study. The related works are listed in Section V, and we conclude in Section VI.

II. Object Teams model and OT/J programming language

In object-oriented collaboration-based applications, objects are interacting within one or more contextual regions called collaborations, and a collaboration can spin several objects. Each object can participate, by exporting part of its behavior, at each spinning collaboration [12] to fulfill its own role in order

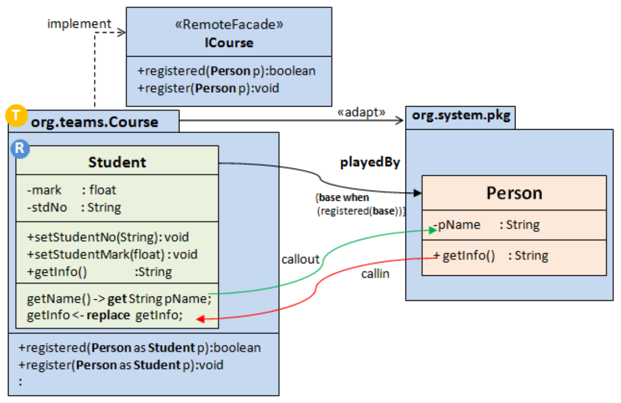
implement
«RemoteFacade»
ICourse
+registered(Person p):boolean
+register(Person p):void
T
org.teams.Course
«adapt»
org.system.pkg
R
Student
-mark : float
-stdNo : String
+setStudentNo(String): void
+setStudentMark(float) : void
+getInfo() :String
getName() -> get String pName;
getInfo <- replace getInfo;
+registered(Person as Student p):boolean
+register(Person as Student p):void
:
playedBy
{base when (registered(base))}
Person
-pName : String
+ getInfo() : String
callout
callin

Fig. 1. Simple OT/J example: **Person** is playing **Student** role within **Course** team.

to complete application functionality. The Object-oriented programming model encapsulates the definition of application objects within modules (e.g. Classes in Java), but fails to provide an adequate module to capture the collaboration.

Observing the intersection between collaborations and application classes, Object Teams model (OT for short) encapsulates collaborations in new modules called *Teams* and captures the intersected behaviors of application classes (now called *base players*) in modules called *Roles*. Considering this modularity structure, a team module is a container of confined roles.

OT/J extends Java by realizing the collaboration role-based design, supports AOP concepts, and integrates them with the object-oriented concepts. Application developers can define teams as first-class modules using the keyword `team`. Inside a team class, roles can be declared as normal Java classes and can be either bounded or unbounded roles. The bounded roles are those roles coupled to base players through the so-called `playedBy` relationship, and the unbounded roles are the rest. The `playedBy` relationship is the only link that allows players to play roles within teams. For example, Fig. 1 shows a role-playing link between **Student** role and its player **Person**.

The relationship `playedBy` involves two types of data-communication channels between a role and its player, namely: Callin Method Bindings (CIMBs) channel, and Call-out Method Bindings (COMBs) channel. A CIMB is an expression that binds a role-level method to a specific player method such that; whenever the player's method is called, the control flow is transferred into a corresponding role instance to execute that role-level method. The point at which control flow must be intercepted is conducted by so-called CIMB modifiers. The OT model presents three modifier types; `before`, `after`, and `replace`. The `before` modifier indicates that the role-level method must be executed before the corresponding player's method starts executing, while `after` modifier implies executing player's method first, and before the control flow leaves that method, it must be dispatched toward the bounded role instance to execute its method. In case of CIMBs with `replace` modifier (called Callin Replacements), the player's method is overridden by the role-level method; i.e. whenever player's method is called, the corresponding role-level method is invoked instead. For example, the CIMB (`getInfo <-` **replace** `getInfo;`), shown in Fig. 1, indicates that whenever `getInfo` method of a base instance of type **Person** is called, the corresponding role-instance's `getInfo` method is invoked.

When the control flow is dispatched by player objects and crosses the boundaries of team instance to execute a CIMB, a valid role instance is picked up (or created if non) to complete the execution via mechanism called *Base Lifting*; which means: pick the corresponding role instance bounded to the caller base object and invoke its desired method.

On the contrary, a COMB is a mechanism to forward the control flow from a role instance into its bounded base instance to invoke a base method on behalf of expected method invocation. In fact, OT/J uses COMBs to realize the *Base Decapsulation* concept; which allows role instances to access their players' attributes and members. For example, the COMB expression (`String getName() -> get String pName;`) declares a new role method called `getName()` and whenever it's invoked, the value of `pName` base's field is returned. To precisely handle COMBs, OT/J uses an implicit operation called *Role Lowering*; which picks the bounded base object up and invoke/access its wanted method/field.

Furthermore, OT/J allows application programmers to control at runtime the effects of callins through Team Activation mechanism; which indicates that all `playedBy` relationships declared in a team class are disabled in an instance of that team unless that instance is implicitly or explicitly activated. Conversely, all `playedBy` relationships of an active team instance become inapplicable if that team is deactivated.

To add more refinement at `playedBy`, OT/J allows the use of conditional expressions, called *Guard Predicates (GPs)*, along with `playedBy` declarations, role-level methods, or CIMB expressions to control their effects at runtime. A special type of GPs called *Base Guard Predicate (BGP)* that comprises one or more base class members (fields or methods) within the expression of condition. For example, Fig. 1 illustrates the BGP (**base when** `(registered(`**base**`))`), which enables only those person instances that are registered in a specific course collaboration playing **Student** roles.

The binding process between base objects and role instances is handled transparently at runtime and triggered either when a specific CIMB is executed for the first time, or due to Explicit Parameter Lifting [11](§2.3.2). For example, the expression (`Person as Student p`) shown in Fig. 1 leads to bind the passed **Person** instance with a role instance once team method `register` is invoked. The first option is implicit and happens once a base method call is intercepted by a CIMB, then the corresponding role instance preserves that base object locally along its life-cycle.

III. A Distributed Model of OT

The extension of OT model into distributed environments is primarily aim at enabling the remote playing of roles. This is to say, we want components and objects of distributed applications be able to play roles remotely preserving the systematic and modular considerations of OT `playedBy` relationship. Regarding OT/J, there is a lack in remote role-playing support due to language-dependent concerns and platform-specific considerations.

On the one hand, OT/J's runtime library (called Object Teams Runtime Environment (OTRE)) requires player classes to be exist at load-time to fulfill the process of bytecode transformation. In fact, OTRE dedicates a special bytecode transformer to inject, at load-time, into players' bytecode all callin interceptions and other supportive code segments according to a prior knowledge of all played roles. On the other, distributed objects in Java are mostly implemented by using interface-based design. In this regard, OT/J does not support a full roles-to-interface binding due to OT/J compiler limitations and type-safety concerns. Indeed, current Object Teams Development Tooling (OTDT) (versions 0.7.x and later under the umbrella of Eclipse) enables binding a role class to interface player but restrict declaring only COMBs.

A. The Concepts of Distributed ObjectTeams/Java

Considering distributed aspects, adopting OT/J concepts for representing aspects as roles, employing them through `playedBy` relationships and control their effects by the BGPs and teams (de)activation (1) improves the modularity and implementation of distributed aspects; which simplifies distributed application development, and (2) allows programmers to dynamically control the applicability of these aspects without shredding the original functionalities of remote base objects.

Basically, we would like to realize remote role-playing and enables the binding of remote players and roles. This involves establishing a communication channel between remote players and the roles to play in order to facilitate remote CIMBs, remote COMBs, and remote BGPs. Practically, the design and implementation of remote role-playing must be conducted by OT disciplines concerning `playedBy` relationship. For example, Role-Confinement rule [11] must be respected to prevent role instances leaving their enclosing team boundaries without necessity.

Conceptually, we define *Remote Team* as the OT/J team class that satisfies one of the following two conditions (or both):

1) If team instances are required to be accessed remotely; i.e. from outside their executing Java Virtual Machine (JVM). In this case, teams are considered as distributed components that comprise a set of services, or as containers of objects .
2) If it encloses at least one role class that is desired to be played by a remote base (i.e. base objects are residing at different host/process than that of roles). As a concept, we call such roles as *Remote Roles*. This way, a modular mechanism can be achieved to implement, and dynamically employ and manage distributed aspects .

A *Remote Base* (or *Remote Player*) is any valid application object that can play one or more remote roles. The significant issue now is establishing precise and accurate bindings between remote roles and their remote bases to achieve remote role-playing. The binding activities must be handled transparently and trigger only due to remote CIMB executions or passing remote base objects through explicit remote-parameters lifting declarations.

It's evident that remote players can play several remote roles within several remote teams. In this concern, remote players need to designate the remote team instances (where their roles are confined) in order to spark remote role-playing. In OT/J, it's the responsibility of OTRE transformers to inject into bases bytecode classes the necessary code segments required to achieve this goal, and the code that represents CIMB channel. While this process is hard to implement in the distributed case, DOT/J distinguishes between remote team designation and remote role-playing (i.e. remote CIMB channel). The first requested to insure that remote bases obtain the correct list of remote teams all the time, while the second represents the actual communication-channel between remote base objects and their role instances. Therefore, to accomplish a flexible remote teams designation, DOT/J enables remote base objects obtaining the latest Remote Teams List (RTL) in a process called *Looking up* and facilitates remote teams appending through *Registration* process.

Respecting remote role-playing, remote bases and remote teams need to contact each other before the binding takes place. In this regard, we adopt the technique of *Provided-Required Interface* to connect remote bases and remote teams. In fact, roles can *directly* access their players' attributes (through COMBs), while the later can only issue CIMB invocations that are manipulated first by remote team which orientates them to a specific role.

B. DOT/J Model Implementation

From a middleware point of view, DOT/J acts as a distributed layer that facilitates a seamless communication for objects participating in remote role-playing. Fig.2 depicts the structure of DOT/J model and illustrates its incorporation with OT/J model. Primarily, DOT/J comprises two main structures, namely: The DOT/J Transformation Library and the DOT/J Runtime Layer (DRTL). Along this section we will demonstrate these structures.

Practically, extending the OT model must regard its fundamentals from one side, and copes with the aforementioned obstacles appropriately from the other. To do so, we divided the extension process into three main stages according to OT/J application development process, namely; static level (or source code phase), load-time level, and runtime level.

1) The static level: The remote role-playing process is a relationship comprises remote base object and a role instance confined within a remote team instance. As remote instances

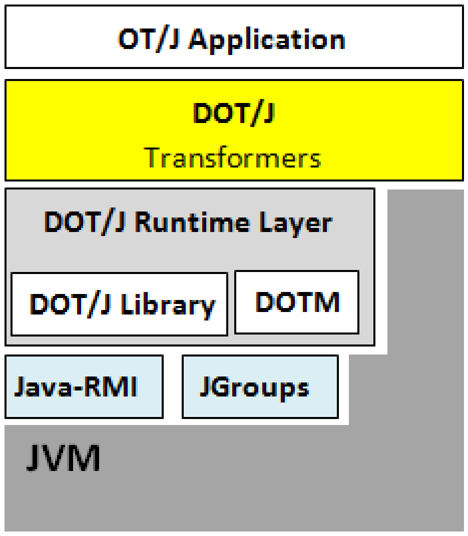

Fig. 2. The structure of DOT/J Model.

are disjointed logically or physically regarding their execution scope, we require their classes to be generated precisely to participate in this remote relationship. Therefore, either we reformulate OT/J's compiler to manipulate remote teams and their enclosed roles according to DOT/J requirements, or reuse the locally generated classes and adjust them to conform with DOT/J disciplines. As the first option implies adding new keywords to OT/J language (e.g. `remotePlayedBy` or `remoteBase`) or compiler directives, the second requires only the designation of classes that participate in remote role-playing relationships. Currently we adopt the second option and leave the first one as future work.

The designation of remote classes is carried out through a process called remote-class *Labeling*. A simple XML file can be used to statically *map* teams, roles, and base classes into DOT/J system through labeling them as remote classes.

2) Load-time level: At this level, the compiled classes of an OT/J application are loaded into JVM for execution. Classes that were labeled at static phase as remote must be prepared with the capability for precise and secure remote role-playing. For example, remote base object should be able to easily look up remote teams that enclosing their roles, and remote teams should be able to register seamlessly.

To equip remote objects (teams and bases) with a precise capability to easily discover each other, and to achieve a transparent binding between remote base objects and role instances, we implement the DOT/J Transformation Library that includes two bytecode transformers; remote-base transformer and remote-team transformer. The first one is dedicated to transform the labeled remote base classes by injecting into their bytecode the necessary code segments that enable their objects looking remote team instances up, handle remote CIMBs precisely, and make them able to serve all remote COMBs accurately.

The other transformer injects into remote team classes the

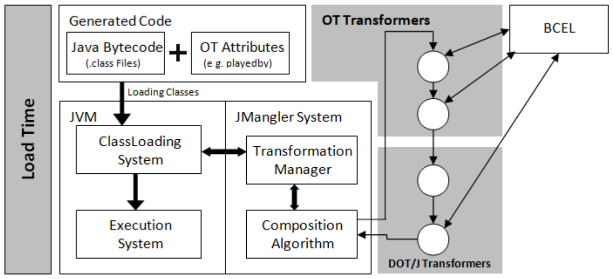

Fig. 3. Integrating DOT/J transformers with OT/J structure.

required code segments that conduct them during registration process, enable them verifying and handling all received remote CIMBs and remote calls, and facilitate COMB invocations and remote BGPs evaluation.

The transformers of DOT/J are using the Byte Code Engineering Library (BCEL) [8] (a toolkit that helps developers to statically analyze Java classes and dynamically transform them at load-time, or create new ones at fly). Fig. 3 illustrates the incorporation between DOT/J transformers and OT/J structure[1]. Note that the output classes of OT/J transformers are the input of DOT/J transformers.

3) The runtime level: Adopting the *Registration-Lookup* strategy imposes the usage of intermediate component between remote team instances and remote base objects. For this purpose, we implement The DOT/J Runtime Layer (DRTL) which comprises two main components: the DOT/J Library and the Distributed Objects and Teams Manager (DOTM). The DOT/J Library includes the necessary packages that insure the accurate and transparent binding of remote team and remote base instances. Particularly, this library involves two domain-specific remote interfaces called the Generic Remote Interfaces (GRIs) that realize the Provided-Required Interface relationship between remote base objects and remote team instances. The first GRI, called `IRemoteBase`, is dedicated to represent a remote façade of remote base objects, and declare a set of generic remote methods that allow remote COMBs invocations and remote BGPs evaluation. All remote base objects must implement this interface to supply a Provided-Interface (DOT/J transformers prepare the complete implementation transparently) .

The second GRI is called `IRemoteTeam` which represents a generic remote façade for all remote team instances, and must be implemented by all remote teams. This GRI is the Provided-Interface required by remote base objects. It declares a set of generic remote methods that allow remote CIMBs invocations and invocations of public remote team methods. Additionally, it comprises DOT/J's special-purpose remote APIs to manage team instances remotely. All remote methods are carried out through Java-RMI [14] middleware system. The DOTM is the core component of DOT/J Runtime

[1]Current OT/J version is mainly use Java Programming Language Instrumentation Services (JPLIS), a mechanism for bytecode instrumentation, and preserves using JMangler [6] class-loader system

system. Remote team instances register themselves into the DOTM records once they are created providing a complete list of remote roles along with remote CIMBs they declare. Respecting remote bases, they *hook* into the DOTM an anchor so that a communication channel is opened between the two parts to send/receive any notifications concerning RTL updates. However, the DOTMs are using JGroups [1], a reliable multicast communication system, to communicate with each other to preserve the RTL consistence and up-to-date, as well as facilitate any recovery processes.

Using the GRIs, DOT/J supports an implicit and transparent binding/lifting of remote base objects/roles. To accomplish the explicit binding/lifting in distributed applications (i.e. through explicit parameter lifting declarations), the DRTL enables the use of so-called *Remote Façades*; which are user-defined Java interfaces that can, in addition to normal methods, define those methods containing explicit parameters lifting declarations in their signatures. At static phase, application developer needs to mark Remote Façades inside the XML files; at both remote team and remote base application parts. For example, the following XML entry declares `ICourse` interface (shown in Fig. 1) as remote façade:

```
<RemoteFacade class="org.teams.ICourse"/>
```

IV. Case study: A Simple Distributed Course Application

Considering the simple OT/J example shown in Fig. 1, we will explain step by step how to implement that application using DOT/J. We assume that the team **Course** application is implemented at host JVM-1, and the base **Person** is implemented at different host, say JVM-2. The following OT/J code fragment illustrates the implementation of **Course** team and the enclosing role **Student**:

```
public team class Course {...
  protected class Student //Role
      playedBy Person
      base when (registered(base))
  {...
  // callin replacement ...
  callin String getInfo()
  { if(!fired(this))   return "STD:"
         +getName()+" No.= " +...;
    return base.getInfo();// proceed
  }
  // CIMBs .. COMBs ..
  getInfo     <- replace getInfo;
  String getName()-> get String pName;
}}
```

A. Labeling Remote Classes

To mark **Person** as remote base class, programmers need only to provide the full-qualified name of that class as follows:

```
<DOTJ>
 <RemoteBases>
   <Base class="org.system.pkg.Person"/>
 </RemoteBases>
 <RemoteFacade class="myPkg.ICourse"/>
</DOTJ>
```

Similarly we can mark **Course** team and **Student** role as remote team and remote aspect, respectively, as follows:

```
 <RemoteTeam class="org.teams.Course">
  <RemoteRole name="Student"/>
  <RemoteFacade class="myPkg.ICourse"/>
 </RemoteTeam>
```

B. Running the example

When creating a new remote team instance, it's automatically registered in the DOTM. Programmers can activate remote teams remotely or *locally* as follows:

```
Course math = new Course("Math 101");
math.activate (Team.ALL_THREAD);
```

At JVM-2, the `math` team instance can be accessed by first contacting the DOTM as follows:

```
1: Person p1 = new Person("A. Odeh");
2: ICourse cMath = (ICourse)
   DOTM.getRemoteTeam("org.teams.Course");
3: cMath.register(p1);//register new student
      :
4: String info = p1.getInfo();
     //remotely deactivated
5: cMath.deactivate(Team.ALL_THREAD);
6: String info2=p1.getInfo();
      :
```

When `getInfo()` method is called (line 4), the control flow is intercepted by a CIMB transparently, then forwarded into `math` instance via its registered GRI, and by passing the GRI of `p1` the binding is accomplished and a role lifting is carried out. The method `getInfo()` of the lifted role instance is invoked and its result returned. Note that `getInfo()` method of role instance issues a remote COMB to get the field `pName` of its bounded base. When `math` team instance is remotely deactivated (line 5) *all* CIMBs become inactive. Therefore, `getInfo()` call at line 6 will execute the local original method of `p1`.

C. Evaluation

In this example, we implement a simple distributed OT/J application using our extending model, DOT/J. The role **Student** is employed transparently to extend the behavior of **Person** class within **Course** collaboration. Most of runtime is consumed in the preparation of DRTL (e.g. loading JGroup protocol staff and joining DOT/J group). The actual runtime for invoking `getInfo()` of instance `p1` while the remote team is activated is 90 ms at the first call, and 3ms (in average) for sequent calls. Our example is implemented using Eclipse IDE v 3.5.2 and OTDT v 1.3.3.

V. Related Works

Mainly, we relate DOT/J to Distributed AOP languages from the perspective of supporting distributed aspects regardless of providing pointcut model.

AWED [7] extends the model of JAsCo[3], an aspect-oriented language tailored for component-based software development. JAsCo allows developers to implement generic Hooks inside aspect modules, and use *Connector* modules to bind hooks and the target components at runtime. AWED makes use of the Connector Registry component to facilitate communication between hosts to serve aspects deployment and remote pointcuts evaluation. Our distributed model uses the DOTM which holds all remote teams, deploy their GRIs (or remote Facades if determined) at all executing nodes. In AWED, aspects must be deployed at all meant hosts, while in DOT/J only remote façades are exposed and aspects remain confined within team instances. Moreover, AWED doesn't support dynamic activation/deactivation of aspects like DOT/J, or the possibility to attach new remote teams dynamically without reloading application components.

JAC [10] is an object-based framework for AOP in Java. Application objects are wrapped so that aspect objects in JAC can be deployed and undeployed dynamically at runtime. Concerning distributed aspects, JAC simulates remote advices by executing local advice on a local copy of aspects; which imposes distributed aspect replication on each host, and adds extra communication efforts to preserve aspects consistency. In DOT/J aspects are enclosed within team instances and advices are executed by the means of remote CIMB execution. Additionally, DOT/J enables the dynamic attachment/detachment of remote teams which provides a dynamic evolution of applications in modular way and requires only remote teams' GRIs to be deployed. The attention is payed in preserving RTL consistency.

DJcutter [9] is an extension of AspectJ. It uses `host` pointcuts, as in AWED, to indicate at which hosts joinpoints must be detected and remote advices must be executed. This technique reduces the transparency of DJcutter. DOT/J provides a high-degree of transparency regarding remote teams detection and remote CIMBs execution. Moreover, DJcutter implements a centralized aspect-server that collects joinpoint information about remote pointcut definitions, and executes related advices locally; i.e. at aspect-server itself. However, the centralized Aspect-server forms a single-point-of failure, as well as it may cause bottleneck problems when large-scale distributed applications are considered.

DyMAC [2] is an aspect-oriented and component middleware framework that uses aspect composition to connect application business-logic to the middleware services. DyMAC is transparently extending the power of aspect composition (joint point model and advice execution) in distributed contexts. DOT/J presents similar approach for handling aspects and detecting CIMBs transparently. The difference between the two approaches is that DyMAC implements a join-point model while DOT/J is only support CIMBs which declare one-to-one method interceptions. Additionally, DOT/J exhibit more flexible and transparent mechanism to declare, deploy and access remote teams through Remote Façades contrary to DyMAC that uses *Component Factory* to instantiate distributed components on-demand and stores them in Distributed Instance Registry then transparently deploy them at hosts.

VI. Conclusions

Distributed AOP languages and models improve the modularity of distributed aspects and distributed-aspect components, and provide several mechanisms to facilitate aspects deployment and remote advice execution. These models lack supporting a dynamic management of distributed aspects like; dynamic activation/deactivation. This paper presents DOT/J, a distributed model that extends into distributed environment the model of OT/J programming language . The model makes use of teams activation/deactivation processes to control aspects applicability at runtime. The separation between base objects and the played roles allows the interception of bases methods without affecting their state consistency or deform their internal structure, i.e. they are not compromised. The model also supports other distributed computing features, such as; distributed collaboration-based application development, distributed components, and modular distributed application adaptability.

References

[1] B. Ban. *JGroups, a toolkit for reliable multicast communication*. http://www.jgroups.org/, 2002

[2] B. Lagaisse, and W. Joosen. *True and transparent distributed composition of aspect-components*. In Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware (Middleware '06), Michi Henning and Maarten van Steen (Eds.). Springer-Verlag New York, Inc., New York, NY, USA, 42-61. 2006

[3] D. Suve, W. Vanderperren, and V. Jonckers, *Jasco: an aspect-oriented approach tailored for component based software development*. In Proceedings of AOSD'3. ACM Press, NY. U.S.A., p:21-29. 2003.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, and et. al. *An Overview of AspectJ*. In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01), Lindskov Knudsen (Ed.). Springer-Verlag, London, UK, 327-353. 2001

[5] G. Kiczales, J. Lamping, A. Mendhekar, and et. al. *Aspect-oriented programming*. In Proceedings of the ECOOP'97. Springer-Verlag LNCS 1241, Finland, p:220-242. 1997.

[6] G. Kniesel, P. Costanza, and M. Austermann, *JMangler - A Framework for Load-Time Transformation of Java Class Files*. Proceedings of IEEE Workshop on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society Press. 2001

[7] L. Navarro, M. Sudholt, W Vanderperren, and et. al. *Explicitly distributed AOP using AWED*. In Proceedings of AOSD'06. ACM, New York, NY, USA, p:51-62. 2006.

[8] M. Dahm. *Byte Code Engineering*. web page: http://jakarta.apache.org/bcel, Dec. 2010

[9] M. Nishizawa, S. Shiba, and M. Tatsubori. *Remote pointcut - a language construct for distributed AOP*. In Proc. of AOSD04. ACM Press, 2004

[10] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin.*JAC: A flexible solution for aspect-oriented programming in Java*. In Proceedings of Reflection01, volume 2192 of LNCS. Springer-Verlag, Sept. 2001

[11] S. Herrmann, C. Hundt, and M. Mosconi. *OT/J Language Definition V1.3*. Technical Universitäte Berlin. http://www.objectteams.org, 2009

[12] S. Herrmann. *Object teams: Improving modularity for crosscutting collaborations*. In Procs. of Net.ObjectDays. Springer, p:248-264. 2002

[13] Spring AOP. http://www.springframework.org/, (Dec. 2010)

[14] Sun Microsystems.*Java Remote Method Invocation Specification V1.5.0*. Sun Microsystems Inc., Santa Clara, California,U.S.A. 2004